Computación y programación en R: Tema 2

David V. Conesa Guillén

GEeitEma

Valencia Bayesian Statistics group

Dept. d'Estadística i Investigació Operativa

Universitat de València

Tema 2: Manejo de datos.

En este tema:

1.- Objetos para el manejo de datos.

2.- Características de los objetos en R: modos y atributos. Datos especiales.

3.- Asignación. Operadores lógicos. Coerción de tipos.

4.- Vectores. Factores. Generación de secuencias regulares. Vectores de índices.

5.- Variables indexadas (arrays).

Tema 2: Manejo de datos.

También:

6.- Matrices. Operaciones con matrices.

7.- Listas.

8.- Los `data.frame` u hojas de datos: las funciones `attach` y `dettach`.

9.- Lectura de ficheros de datos.

10.- Importar y exportar datos de otros programas.

1.- Objetos para el manejo de datos.

- `R` es un lenguaje orientado a objetos: las variables, datos, funciones, resultados, etc., se guardan en la memoria activa del ordenador en forma de objetos con un nombre específico.
- Podemos modificar o manipular estos objetos con operadores (aritméticos, lógicos y comparativos) y funciones (que a su vez son objetos).
- Las estructuras de datos más usuales: variables, vectores, matrices, factores, variables indexadas, cadenas de caracteres, listas y data frames.
- Existen otros como series temporales y es posible crear nuevas estructuras de datos.
- Recordar el funcionamiento de `objects()`, `ls()` y `rm()`.

2.- Características de los objetos en R: modos y atributos.

Los objetos están compuestos de elementos. Los elementos más simples, las variables, pueden ser:

Tipos o modos de datos

- `numeric`: número real con doble precisión. Los podemos escribir como enteros (3, -2), con fracción decimal (3.27) o con notación científica (3.12e-47).
- `complex`: números complejos de la forma `a+bi`.
- `character`: Cadenas alfanuméricas de texto.
- `logical`: variables lógicas. Puede ser `TRUE` o `FALSE`.

Ejemplo

```
nombre<-"Luis"
varon<-TRUE
edad<-23
estatura<-1.77
```


Datos especiales: `NA`'s.

- En algunos casos las componentes de un objeto pueden no ser completamente conocidas.
- Cuando un elemento o valor es "not available" le asignamos el valor especial `NA`.
- En general una operación con elementos `NA` resulta `NA`, a no ser que mediante una opción de la función, podamos omitir o tratar los datos faltantes de forma especial.
- La opción por defecto en cualquier función es `na.rm=FALSE` (que indica que NO elimina los `NA`), que da como resultado `NA` cuando existe al menos un dato faltante.
- Con la opción `na.rm=TRUE`, la operación se efectúa con los datos válidos.

Datos especiales: NA's.

Ejemplo

`x<-NA`
: Asignar NA a la variable x

`x+1`
: Observar que el resultado es también un NA.

`y<-c(x,3,5,x)`
: Asignamos al vector y dos NA y dos números.

`mean(y)`
: Calcula la media teniendo en cuenta los NA's.

`mean(y,na.rm=TRUE)`
: Calcula la media SIN tener en cuenta los NA's.

Datos especiales: Inf y NaN's.

- En la mayoría de los casos, no debemos preocuparnos de si los elementos de un objeto numérico son enteros, reales o incluso complejos. Los cálculos se realizarán internamente como números de doble precisión, reales o complejos según el caso.
- Para trabajar con números complejos, deberemos indicar explícitamente la parte compleja.
- En determinadas ocasiones los cálculos realizados pueden llevar a respuestas con valor infinito positivo (representado por R como Inf) o infinito negativo (-Inf).
- Es posible realizar y evaluar cálculos que involucren Inf.
- Sin embargo, a veces, determinados cálculos llevan a expresiones que no son números (representados por R como NaN's, del inglés 'not a number').

Datos especiales: `Inf` *y* `NaN`*'s.*

Ejemplo

`sqrt(-17)`
: Produce un "Warning message" avisando que es un `NaN`.

`sqrt(-17+0i)`
: Calcula el resultado y devuelve un número complejo.

`x<-5/0`
: Asignamos a `x` un valor infinito.

`exp(-x)`
: Devuelve el valor 0.

`exp(x)-exp(x)`
: Devuelve `NaN`: es una indeterminación.

Clasificación de los objetos en `R`*.*

- Los objetos se pueden clasificar en dos grandes grupos:
  - *atómicos*: todos los elementos que los componen son del mismo tipo (o modo), como por ejemplo los vectores, matrices, series temporales.
  - *recursivos*: pueden combinar una colección de otros objetos de diferente tipo (o modo), como son los data.frame, listas.
- Existen otras estructuras recursivas, p.e.:
  - El modo `function` está formado por las funciones que constituyen `R`, unidas a las funciones escritas por cada usuario. Las trataremos en un tema posterior.
  - El modo `expression` corresponde a una parte avanzada de `R` que trataremos más adelante al presentar las *fórmulas* en el tema de modelos estadísticos.
- Recordar además que es posible crear nuevas estructuras.
- Utilizando la función `str(objeto)` podemos obtener información sobre su estructura.

Atributos de los objetos.

- Los atributos de un objeto suministran información específica sobre el propio objeto.
- El modo o tipo de un objeto es un caso especial de un atributo de un objeto. Con el modo de un objeto designamos el tipo básico de sus constituyentes fundamentales.
- Los atributos de un objeto suministran información específica sobre el propio objeto. **Todos** los objetos tienen dos atributos intrínsecos: el modo y su longitud.
- Las funciones `mode(objeto)` y `length(objeto)` se pueden utilizar para obtener el modo y longitud de cualquier estructura.

Ejemplo

```
x<-c(1,3)
mode(x) # Devuelve el tipo numeric
length(x) # Devuelve la longitud
```


Atributos de los objetos.

- Mediante `attributes(objeto)` podemos obtener una lista de los atributos **no intrínsecos** y con `attr(objeto,atributo)` podemos usar el atributo seleccionado (p.e. para asignarle un valor).
- Los atributos son distintos según el tipo de objeto. Una pequeña lista de atributos es la siguiente:

Atributos de cada tipo

Atributo	Tipos
mode	Todos
storage.mode	Todos los datos de modo numérico
length	Todos
names	Vectores y listas
dim	Matrices y arrays
dimnames	Matrices y arrays
tsp	Series temporales
levels	Factores

Modificación de la longitud de un objeto

- Un objeto, aunque esté vacío, tiene modo.

Ejemplo

`v <- numeric()` almacena en `v` una estructura vacía de vector numérico. `character()` es un vector de caracteres vacío, y lo mismo ocurre con otros tipos.

- Una vez creado un objeto con un tamaño cualquiera, pueden añadirse nuevos elementos sin más que asignarlos a un índice que esté fuera del rango previo.

Ejemplo

`v[3] <- 17` transforma `v` en un vector de longitud 3, (cuyas dos primeras componentes serán `NA`).

- Esta regla se aplica a cualquier estructura, siempre que los nuevos elementos sean compatibles con el modo inicial de la estructura.

3.- Asignación. Operadores lógicos. Coerción de tipos.

- Recordar que la función principal para definir un objeto es a través de sus componentes, con la función `c()`, mediante el comando más importante en `R` que es `<-` el de la asignación.
- Las asignaciones pueden realizarse también con una flecha apuntando a la derecha, realizando el cambio obvio en la asignación.

Ejemplo

`x <- 3` es equivalente a `3 -> x` pero no a `x < - 3`

- La asignación puede realizarse también mediante la función `assign()`.

Ejemplo

`x<-c(1,3)` es equivalente a `assign("x", c(1, 3))`

- En muchos contextos el símbolo $=$ puede también utilizarse indistintamente para asignar un objeto.

Asignación.

- Si una expresión se utiliza como una orden por sí misma, su valor se imprime y se pierde. Así pues, la orden `1/x` simplemente imprime el inverso de lo que sea el objeto `x` sin modificar su valor.
- Podemos crear vectores con valores iniciales, FALSE, 0,0+0i, " ", mediante la función que indica el tipo de dato y entre paréntesis el numero de elementos a crear.

Ejemplo

```
x1<-logical(4)
```
Inicializa un vector de longitud 4 cuyos elementos son FALSE

```
x2<-numeric(4); x3<-complex(4)
```
Inicializa (a 0) vectores numéricos de longitud 4

```
x4<-character(4)
```
Inicializa un vector de caracteres de longitud 4

Operadores lógicos.

- Los elementos de un objeto de tipo lógico tienen dos posibilidades, FALSE o TRUE. Se pueden abreviar en F y T respectivamente.
- Los objetos lógicos son generalmente fruto de una comparación.

Ejemplo

Si `x <- 3`
`y <- x>12` produce un objeto lógico de longitud la de `x` con valor `F` o `V` según se cumpla o no la condición (en este caso `F`)

- Las comparaciones que dan un resultado lógico son: $<, <=, >, >=, ==, != .$
- Los objetos lógicos se pueden utilizar con la aritmética ordinaria, en cuyo caso son transformados en vectores numéricos, `FALSE` se convierte en 0 y `TRUE` en 1.

Operadores lógicos.

La aritmética entre objetos lógicos se puede llevar a cabo con los siguientes operadores lógicos:

Operadores lógicos

Operador	Operación
!x	Negación de x. Los T los convierte en F y viceversa.
x&y	Intersección, operador lógico y: T y T da T, otra comparación da F
x|y	Unión, operador lógico o: F y F da F, otra comparación da T.
`xor(x,y)`	Exclusivo OR, `xor(T,F)==T`, otra comparación da F.
`all`	Para una secuencia de argumentos lógicos, `all` devuelve el valor lógico que indica si todos los elementos son TRUE.
`any`	Para una secuencia de argumentos lógicos, `any` devuelve el valor lógico que indica si algún elemento es TRUE.

Operadores lógicos.

Ejemplo

```
x<-c(T,T,F,F)
y<-c(T,F,T,F)
x&y # produce TRUE FALSE FALSE FALSE
x|y # produce TRUE TRUE TRUE FALSE
xor(x,y) # produce FALSE TRUE TRUE FALSE
any(x) # produce TRUE
all(x) # produce FALSE

z<- c(NA, T, F)
?outer # ayuda sobre el producto exterior de dos vectores
outer(z,z,"&")
outer(z,z,"|")
```

Coerción de tipos.

- La mayoría de las funciones producen un error cuando el tipo de datos que esperan no coincide con los que ponemos en los argumentos.
- Tenemos dos posibilidades:
  - comprobar el tipo de datos utilizando funciones `is.algo()`, que nos responde con un valor lógico,
  - o forzar al tipo de datos deseados **coercionando**, para lo cual podemos utilizar funciones del tipo `as.algo()`, que fuerzan el tipo de datos.
- Para conocer si un elemento es `NA`, la comparación lógica ==, no puede funcionar ya que `NA` es la nada. Utilizamos para ello la función `is.na()`, que nos responde con un valor lógico: `TRUE` para los elementos del vector que son `NA`.
- De manera similar, podemos utilizar `is.finite()`, `is.nan()`, etc.

Coerción de tipos.

Lista de los tipos más importantes que se pueden comprobar o forzar

Tipo	Comprobación	Coerción
array	is.array()	as.array()
character	is.character()	as.character()
complex	is.complex()	as.complex()
double	is.double()	as.double()
factor	is.factor()	as.factor()
integer	is.integer()	as.integer()
list	is.list()	as.list()
logical	is.logical()	as.logical()
matrix	is.matrix()	as.matrix()
NA	is.na()	-
NaN	is.nan()	-
NULL	is.null()	as.null()
numeric	is.numeric()	as.numeric()
ts	is.ts()	as.ts()
vector	is.vector()	as.vector()

Ejemplo

```
x<-c(1:10)

is.numeric(x)
```
resulta en TRUE

```
is.vector(x)
```
resulta en TRUE

```
is.complex(x)
```
resulta en FALSE

```
is.character(x)
```
resulta en FALSE

```
x<-as.character(x)
```
fuerza el vector `x` a tomar los valores ("1" "2" "3" "4" "5" "6" "7" "8" "9" "10")

4.- Vectores. Factores. Generación de secuencias regulares. Vectores de índices.

Características de los vectores

- Extienden los tipos básicos
- Concatenación de elementos
- Todos los elementos del vector han de ser del mismo tipo
- Se crean con la función `c()`
- Se puede usar `vector[5]` para acceder al quinto elemento.

Ejemplo

```
nombre<-c("Luis", "María") # Vector de caracteres
edad<-c(23, 24)
varon<-c(TRUE, FALSE); adult <- edad>18
estatura<-c(1.77, 1.64)
estatura[2] #Devuelve 1.64
```

Factores/Variables categóricas.

- Un factor es un vector utilizado para especificar una clasificación discreta de los elementos de otro vector de igual longitud.
- En R existen dos tipos de factores:
  - No ordenados (nominales): No existe jerarquía entre ellos (p.e., colores)
  - Ordenados (ordinales): Existe jerarquía entre ellos (p.e., grupos de edad)
- Se pueden crear a partir de un vector numérico con las funciones `as.factor()`, `as.ordered()` o con la función `gl()`.
- También a partir de un vector de caracteres utilizando `factor()`.
- Las etiquetas se asignan con `levels()`.

Ejemplo

```
f<-as.factor(c(1,2,3,1,2,1,1,3,2)) #Factor con 3 categorías
levels(f)<-c("Bajo", "Medio", "Alto")
ford<-as.ordered(f) # Factor ordenado
```


Vectores de caracteres: la función `paste()`.

- R tiene algunos vectores de caracteres predefinidos: `LETTERS`, `letters`, `month.name` y `month.abb`.
- La función `paste()` une todos los vectores de caracteres que se le suministran y construye una sola cadena de caracteres. Muy útil en gráficas.
- También admite argumentos numéricos, que convierte inmediatamente en cadenas de caracteres.
- En su forma predeterminada, en la cadena final, cada argumento original se separa del siguiente por un espacio en blanco, aunque ello puede cambiarse utilizando el argumento sep="cadena", que sustituye el espacio en blanco por cadena, la cual podría ser incluso vacía.

Ejemplo

`labs <- paste(c("X","Y"),1:10,sep="")` almacena, en `labs`,
`c("X1","Y2","X3","Y4","X5","Y6","X7","Y8","X9","Y10")`

Generación de secuencias regulares.

- R tiene grandes facilidades para generar vectores de secuencias de números.
- El operador más usual es : que genera una secuencia desde:hasta en incrementos (o decremento si hasta es menor que desde) de uno.
- Este operador tiene la prioridad más alta en una expresión.
- La función seq() permite generar sucesiones más complejas. Dispone de varios argumentos (from, to, by, length). Existen funciones similares que realizan los cálculos más rápidamente (muy útiles a la hora de programar). Chequear la ayuda ?seq.
- Una función relacionada es rep(), que permite duplicar un objeto de formas diversas. Su forma más sencilla es rep(x, times=5) que produce cinco copias de x, una tras otra.
- La función sequence(nvec) genera secuencias de vectores concatenados. nvec es un vector de enteros que especifica los límites superiores de las secuencias que se generan. Todas comienzan en uno y se van concatenando en un vector resultante.

Generación de secuencias regulares.

Comprobar las siguientes expresiones:

Ejemplo

```
pi:1
seq(0,1,length=10)
seq(3) # equivale a 1:3 y a seq(1,3)
seq(1,5,by=0.5)
-1:1/0
rep(1:4,2)
rep(1:4,c(2,3,1,2))
rep(1:4,c(2,2)) # error, tamaños no coinciden
sequence(c(2,3))
```

Extrayendo muestras de un vector.

- R nos permite seleccionar muestras de un vector utilizando la función `sample(x,size,replace=FALSE,prob)`:
- Obtiene una muestra de tamaño size de x, con o sin reemplazamiento, pudiendo tener los elementos de x probabilidades distintas a la *uniforme* (por defecto). Si x tiene longitud 1, se considera el vector `1:x`.

Ejemplo

```
x<-c(1:10)
sample(x,3)
sample(x) # Permutaciones de los elementos de x
y<-sample(5:15,5)
y
```


Vectores de índices.

- R nos permite seleccionar subconjuntos de elementos de un vector añadiendo al nombre del vector *un vector índice* entre corchetes `[]`.
- El vector índice puede ser de tres tipos distintos:
  1. Un vector lógico: en este caso el vector índice debe ser de la misma longitud que el vector del cual queremos seleccionar elementos. Los valores correspondientes a `TRUE` en el vector índice son seleccionados, los correspondientes a `FALSE` omitidos.
  2. Un vector de números enteros: en caso de ser positivos, los correspondientes elementos indicados en el índice son concatenados, mientras que si son negativos los valores especificados son omitidos. No hace falta que sea de la misma longitud que el vector de donde seleccionamos.
  3. Un vector de nombres. Esta posibilidad se aplica cuando el objeto tiene el atributo `names` que identifica sus componentes. La función `names` permite añadir etiquetas (o nombres) a un vector numérico.
- La función `which()` genera un vector numérico con las posiciones seleccionadas.

- Vectores índice lógicos:

Ejemplo

```
x<-c(0,NA,1,2)
y<-x[!is.na(x)]
(x+1)[!is.na(x) & x>0] -> z
```

- Vectores de números enteros:

Ejemplo

```
x<-letters[c(13,1,18,9,15)]
paste(letters[c(13,1,18,9,15)],collapse="")
x[-(4:5)]
```

- Vector de nombres:

Ejemplo

```
frutos<-c(5,10,1,20)
names(frutos)<-c("pera","banana","melon","naranja")
postre<-frutos[c("pera","melon")]
```


Más sobre vectores de índices.

Ejemplo

```
x<-0:10
sum(x)
length(x)
sum(x<5)
x[x<5]
sum(x[x<5])
length(x[x<5])
sum(x*(x<5))
sum(which(x<5))

z<-numeric(10)
id<-which(x<5)
z[id] <- x[x<5]
z[which(x<5)] <- x[x<5] # equivalente al anterior (¿mejor?)
```

5.- Variables indexadas (arrays).

- Una *variable indexada* (`array`) es un objeto con elementos todos del mismo tipo con un atributo adicional (`dim`) el cual a su vez es un vector numérico de dimensiones (números enteros positivos) formado por varios índices. Vectores y matrices son casos particulares.
- Los elementos del vector de dimensiones indican los límites superiores de los índices. Los límites inferiores siempre valen 1.
- Existen dos maneras de crear una variable indexada:
  1. Un vector puede transformarse en una variable indexada cuando se asigna un vector de dimensiones al atributo dim.

Ejemplo

```
z<-numeric(1500); dim(z) <- c(3,5,100)
```

  2. Utilizando la función `array(vector de datos,vector de dimensiones)`

Ejemplo

```
h<-numeric(24); Z <- array(h, dim=c(3,4,2))
```


Más sobre variables indexadas (arrays).

- Al igual que en Fortran, la creación de la variable indexada sigue la regla de que el primer índice es el que se mueve más rápido y el último es el más lento.

Ejemplo

Si se define una variable indexada, a, con vector de dimensiones `c(3,4,2)`, la variable indexada tendrá 3x4x2 = 24 elementos que se formarán a partir de los elementos originales en el orden `a[1,1,1]`, `a[2,1,1]`, ..., `a[2,4,2]`, `a[3,4,2]`.

- Para referirnos a un elemento de un array daremos el nombre de la variable y, entre corchetes, los índices que lo refieren, separados por comas.

Ejemplo

`a[1,2,1]`, ..., `a[2,4,2]`, etc.

Más sobre variables indexadas (arrays).

- Podemos referirnos a una parte de un array mediante una sucesión de vectores índices, teniendo en cuenta que si un vector índice es vacío, equivale a utilizar todo el rango de valores para dicho índice.

Ejemplo

a[2,,] es una variable indexada 4x2, con vector de dimensión c(4,2) cuyos elementos son (a[2,1,1], a[2,2,1], a[2,3,1], a[2,4,1], a[2,1,2], a[2,2,2], a[2,3,2], a[2,4,2]) en ese orden.

a[,,] equivale a la variable completa; coincide con a.

- Si especificamos una variable indexada con un solo índice o vector índice, sólo se utilizan los elementos correspondientes del vector de datos, y el vector de dimensión se ignora.

Ejemplo

a[2] sería el elemento 2 del vector a, es decir el a[2,1,1] del array

6.- Matrices. Operaciones con matrices.

- Las matrices son un caso particular de array con dos dimensiones: un objeto con un atributo adicional el cual a su vez es un vector numérico de longitud 2, que define el número de filas y columnas de la matriz.
- Todos los elementos deben ser del mismo tipo.
- Una matriz se define con el comando `matrix()` especificando el número de filas y columnas o asignando la `dim` a un vector.
- Recordar que la matriz se crea por columnas, aunque con la opción `byrow=TRUE` lo hace por filas.
- Podemos asignar nombres a las filas y columnas con el atributo `dimnames`.
- Las funciones `is.matrix()` y `as.matrix()` comprueban o fuerzan el carácter de matriz de un objeto.

Más sobre matrices.

- La situación más sencilla es la de seleccionar unas determinadas filas con un vector y unas columnas con otro vector, siguiendo las reglas comentadas en los arrays.
- R tiende a reducir complejidad: al seleccionar una columna, el objeto resultante es un vector y no una matriz. Para mantener las dimensiones, añadir `drop=F`.
- Podemos seleccionar elementos de una matriz poniendo entre corchetes otra matriz, de las mismas dimensiones que la original, con valores lógicos que indicarán con TRUE los elementos a considerar.
- Cuando se realizan operaciones que mezclan variables indexadas (matrices, p.e.) y vectores conviene tener en cuenta que si asignamos un vector más corto a una matriz, se extiende repitiendo sus elementos (lo que se denomina reciclado) hasta alcanzar el tamaño deseado.

Más sobre matrices.

Ejemplo

```
x<-c(1:6)
dim(x)<-c(2,3)
dimnames(x)<-list(c("Fila1","Fila2"),c("Col1","Col2","Col3"))
ejema<-matrix(1:12,ncol=3,byrow=T,
            dimnames=list(letters[1:4],LETTERS[1:3]))
ejema[1,1]
ejema[,c(2,3)]
ejema[,c(-1,-3),drop=F]
sel<-matrix(rep(c(T,F),6),4,3)
ejema[sel]
```

Más sobre matrices.

Funciones útiles para trabajar con matrices

Función	Utilidad
ncol(x) nrow(x) t(x) cbind(...) rbind(...) diag(x) col(x) row(x)	Número de columnas de x. Número de filas de x. Transpuesta de x Combina secuencias de vectores/matrices por col's. Combina secuencias de vectores/matrices por filas. Extrae diagonal de matriz o crea matriz diagonal. Crea una matriz con elemento ij igual al valor j Crea una matriz con elemento ij igual al valor i
apply(x,margin,FUN,)	Aplica la función FUN a la dimensión especificada en margin 1 indica filas, 2 indica columnas. NB.
outer(x,y,fun="*") otra forma x %o %y	Para dos vectores x e y , crea una matriz A[i,j]=FUN(x[i],y[j]). Por defecto crea el producto externo.

Más sobre matrices.

Algunas operaciones con matrices

Función	Utilidad
x %* %y crossprod(x,y=x) cov(x,y=x,use="all.obs") cor(x,y=x,use="all.obs")	Multiplicación de matrices Idem que t(x) %* %y, pero más rápida Matriz de varianzas-covarianzas Matriz de correlaciones
scale(x,center=,scale=)	Resta a las columnas la media si center=TRUE, Si center es un vector, resta dichos valores. Idem para scale ,luego de centrar, pero divide por la desviación típica si scale=TRUE o los valores asignados si es un vector.
chol(x)	Descomposición de Choleski.
solve(a,b,tol=1e-7)	Resolución de la ecuación a%*%x=b. tol=tolerancia para detectar dependencias lineales en las columnas de a
eigen(x) sdv(x)	Cálculo de valores y vectores propios. Descomposición en valores singulares.

Más sobre matrices.

Ejemplo

```
x<-matrix(1:6,2,3)
x[,2]
x[1,1:2]
y<-matrix(1:6,3,2)
y[3,] ; y[3]
ncol(x); nrow(y)
t(x)
cbind(1,x)
cbind(1:3,1:6)
diag(x) # no es necesario que la matriz sea cuadrada
apply(x,1,sum)
```


7.- Listas.

- Una lista es un objeto consistente en una colección ordenada de objetos que se suelen llamar componentes.
- No es necesario que los componentes sean del mismo modo, ni de la misma longitud: una lista puede estar compuesta de, por ejemplo, un vector numérico de tamaño 2, un valor lógico, un vector de tamaño 3, una matriz y una función.
- Se construyen con la función `list()` o concatenando otras listas.
- Son una parte importante de la programación de funciones en R.
- Los componentes siempre están numerados y pueden ser referidos por dicho número, o por su nombre (si lo tiene, por defecto no lo tiene).
- Debemos ir con mucho cuidado al seleccionar partes de la lista:
  - La selección de elementos se hace con doble corchete o con el nombre del elemento precedido del símbolo del dólar.
  - Pero si utilizamos el corchete simple estamos considerando una sublista (de menos componentes) de la lista.

Más sobre listas.

Ejemplo

```
ejemplolista <- list(nombre="Pedro", casado=T,
       esposa="María",no.hijos=3, edad.hijos=c(4,7,9))
ejemplolista
ejemplolista[5]
is.vector(ejemplolista[5]); is.list(ejemplolista[5])
ejemplolista[[5]]
is.vector(ejemplolista[[5]]); is.list(ejemplolista[[5]])
ejemplolista[[5]][2]
ejemplolista[[5]]
ejemplolista$casado
ejemplolista$nombre
is.recursive(ejemplolista)
is.atomic(ejemplolista) # vemos el tipo de objeto
listamasgrande <- c(ejemplolista,list(edad=40))
```


8.- Los `data.frame`

- Las bases de datos en estadística son, habitualmente de la forma:

	Covariables			
individuo	anyos	implante	edad	sexo
1	1.3	2	22	H
2	0.4	2	21	M
3	1.1	2	34	H
4	2.3	1	42	H
5	3.1	3	17	M
6	1.3	1	43	H

- R organiza este tipo de información en objetos del tipo `data.frame`, un caso particular de `lista`.
- Los `data.frame` son apropiados para describir “matrices de datos” donde cada fila representa a un individuo y cada columna una variable, cuyas variables pueden ser numéricas o categóricas.

8.- Hojas de datos o data.frame

- Los data.frame se crean con la función `data.frame()`:

Ejemplo

```
datosimp <- data.frame(anyos=c(1.3,0.4,1.1,2.3,3.1,1.3),
     tipo=c(2,3,3,1,3,1),edad=c(22,21,34,42,17,43),
     sexo=c("H","M","H","H","M","H"))
```

- Las componentes deben ser vectores (numéricos, carácter o lógicos), factores, matrices numéricas u otros data.frames.
- Puesto que un vector representará una variable del banco de datos y las columnas de una matriz representarán varias variables de ese mismo banco, la longitud de los vectores debe ser la misma y coincidir con el número de filas de las matrices.
- Los datos que no son numéricos, la estructura de data.frame los considera factores, con tantos niveles como valores distintos encuentre, tal y como ocurre en el ejemplo que estamos considerando.

Las funciones attach() y detach().

- Para trabajar con las variables del banco de datos, podemos utilizar la notación estándar de las listas, $nombre o [[ ]], pero resulta más natural emplear simplemente el nombre de la columna.
- El problema es que `R` guarda el nombre del data.frame pero no sus variables. Para poderlas utilizar con su nombre como vectores, tenemos que utilizar la función `attach(nombre de data.frame)`. La operación inversa se realiza con la función `detach()`.
- En cualquier momento es posible llevar a cabo un filtrado de datos utilizando expresiones lógicas (por ejemplo, para seleccionar de un banco de datos sólo los que son mayores de 65 años) y vectores índices.
- En el caso de los data.frames es posible seleccionar filas y columnas deseadas, y crear con ellas otros `data.frame` con las características deseadas con la función `subset()`.

Funcionamiento de `attach()`, `detach()` y `subset()`.

Ejemplo

```
datosimp <- data.frame(anyos=c(1.3,0.4,1.1,2.3,3.1,1.3),
     tipo=c(2,3,3,1,3,1),edad=c(22,21,34,42,17,43),
     sexo=c("H","M","H","H","M","H"))
anyos
attach(datosimp); anyos
detach(datosimp); anyos
datos.hombre.filtrados <- datosimp$anyos[datosimp$sexo=='H']
mas.peq <- subset(datosimp,anyos<1,select=c(edad,sexo))
```


Transformaciones.

- En cualquier momento, podemos modificar o manipular cualquier objeto con operadores (aritméticos, lógicos y comparativos) y funciones. Así, por ejemplo, si una variable no es normal, puedo transformala para conseguir dicha normalidad.
- ¡Cuidado! Cualquier modificación que hagamos con las variables de un banco de datos "adjuntado" (attached) no afectarán al propio `data.frame`.
- Si queremos modificar el contenido interno de los data.frames podemos utilizar la función `transform()`. En concreto podemos:
  - Transformar variables existentes (para conseguir normalidad, por ejemplo).
  - Crear nuevas variables mediante la transformación de variables existentes.

Recodificaciones.

- Mediante la función `recode()` de la librería `car` es posible llevar a cabo recodificaciones de otras variables.
- Debemos especificar cuál es la variable a recodificar, cuáles son los criterios de recodificación y si deseamos que la nueva variable resultante de la recodificación sea un factor o no.
- ¡Cuidado! Las recodificaciones tampoco afectarán al propio `data.frame`.

Funcionamiento de transformaciones y recodificaciones.

Ejemplo

```
attach(datosimp)
edad.final <- edad + anyos # no afecta al data.frame
datosimp.1 <- transform(datosimp, edad.final = edad+anyos) # si afecta
datosimp.2 <- transform(datosimp.1, edad = edad+1) # altera la variable
install.packages("car"); library(car) # Instalar y cargar paquete
edc<-recode(edad," 15:25='joven'; 26:65='adulto' ", as.factor.result=T)
sexo.cod<-recode(sexo," 'H'=0 ; 'M'=1 ", as.factor.result=TRUE)
```

9.- Lectura de ficheros de datos.

- Hasta ahora hemos visto cómo introducir datos interactivamente en forma de objetos, aunque habitualmente los datos nos vienen proporcionados en archivos externos.
- Como los requisitos de lectura de `R` son bastante estrictos, muchas veces conviene modificar el archivo de datos externo con otros editores de texto (como `WinEdt`, etc.).
- Para asignar un banco de datos a un `data.frame` se utiliza la función `read.table()`. La forma más sencilla es con un fichero en el que:
  - La primera línea del archivo debe contener el nombre de cada variable de la hoja de datos.
  - En cada una de las siguientes líneas, el primer elemento es la etiqueta de la fila, y a continuación deben aparecer los valores de cada variable. Si no, `R` asigna automáticamente etiquetas a las filas.

Ejemplo

```
ratas<- read.table(file="F:/Datos/ratas.txt", header=T)
```


La función `scan()`.

- La función `scan()` es más genérica que la anterior, y vale para asignar cualquier tipo de objetos (vectores, matrices, listas, etc.).
- Si el segundo argumento es un sólo elemento, todos los elementos del archivo deben ser del tipo indicado y se leen en un sólo vector.

Ejemplo

En su uso más básico la función `scan()` permite introducir los valores interactivamente.

```
x<- scan()
1:
```

En el prompt nos va pidiendo valores que se van incorporando al vector hasta que pulsamos intro y acaba. Probar a introducir 1, 5, 8 y 3.

Ejemplo

Supongamos que el fichero "entrada.txt" contiene los datos de tres vectores, de igual longitud, el primero tipo carácter y los otros dos tipo numérico, escritos de tal modo que en cada línea aparecen los valores correspondientes de cada uno de ellos:

```
entrada<- scan("entrada.txt", list("",0,0))
```

Podemos referirnos a los vectores con:

```
etiqueta <- entrada[[1]]; x <- entrada[[2]]; y <-
entrada[[3]]
```

También podríamos haber utilizado:

```
entrada <- scan("entrada.txt", list(etiqueta="",x=0,y=0))
```

lo que nos permitiría utilizar la notación $ para referirnos a los vectores:

```
etiqueta <- entrada$etiqueta; x <- entrada$x; y <- entrada$y
```


Otras formas de leer y guardar datos.

- La función `save(objetos, list = character(0), file = "nomfich.ext", ascii = FALSE)` nos permite guardar los *objetos* que queramos en el *nomfich.ext*.
- `load()` nos permite leer datos de un fichero binario que contiene los objetos de `R` previamente guardados con `save()`.
- Como vimos en el primer tema, cuando cerramos `R` existe la posibilidad de guardar todos los objetos de la sesión de trabajo en archivos `.RData`.
- En realidad, al grabar toda la imagen `R` está aplicando la función `save.image()`, un caso particular de la función `save(list=ls(),file=".Rdata")`.
- Al cargar los datos desde el menú `Archivo>cargar área de trabajo` o hacer doble click sobre el fichero `.RData` en el fondo estamos utilizando la función `load()`.

Otras formas de leer y guardar datos.

- Aunque no es muy elegante, siempre es posible guardar el flujo de comandos de `R` que contiene la definición de un `data.frame` y luego recuperarlo con la función `source()` que vimos en el primer tema.
- `data()` lista los bancos de datos disponibles en el package *datasets*, mientras que `data(package = .packages(all.available = TRUE))` lista todos los disponibles en las librerías que hayamos cargado anteriormente.
- La función `data(nom.banco.datos)` carga el banco de datos para que podamos utilizarlo.
- Existen dos funciones más para guardar datos:
  1. `write()` que guarda en un fichero texto vectores y matrices que luego se pueden leer con `scan()` y
  2. `write.table()` que guarda también en fichero texto los `data.frame` para leerlos luego con `read.table()`.

10.- Importar y exportar datos de otros programas.

- La gran aceptación de `R` está haciendo cada vez más fácil el importar ficheros de otros programas.
- La librería `foreign` contiene funciones para importar y exportar datos en el formato de otros programas.
- Utilizando `help(package="foreign")` podemos encontrar una lista de funciones y formatos soportados de otros programas estadísticos (SPSS, SAS, STATA, Minitab, SPlus, EpiInfo, Systat) y numéricos (Octave).

Ejemplo

```
library(foreign) # Cargamos el paquete
datos<-read.spss(file="glucosa.sav", to.data.frame=TRUE)
str(datos)
```

Importar y exportar datos de otros programas.

- Existen otras opciones (ver el manual R-data incluido en el material y la web `http://cran.r-project.org/`) para conectar R con diferentes bases de datos (Access, Excel, MySQL, dBase, etc.).
- La idea es utilizar lo que se conoce como "Open DataBase Connectivity" (ODBC), un estándar de acceso a bases de datos utilizado por la mayoría de bases de datos existentes.
- A través de ODBC, un sistema puede conectarse a cualquier base de datos (que esté local en nuestro ordenador o incluso que esté remota) que soporte ODBC sin necesidad de conocer sus características específicas. Para ello, el desarrollador de la base de datos es el encargado de crear el driver de ODBC necesario para que su base de datos pueda ser utilizada desde ODBC.
- En el caso de R, existe una librería llamada `RODBC` que permite conectarse mediante un ODBC y desarrollar consultas contra bases de datos.

Caso particular: importar datos de Excel

La librería `RODBC` permite conectarse mediante un ODBC a un fichero excel.

Comandos tipo para conectar con Excel

```
library(RODBC) # Cargamos el paquete
conexion<-odbcConnectExcel("dedos.xls")
Datos<-sqlQuery(channel=conexion,"select * from [Hoja1$]")
close(conexion)
```

- La segunda línea establece una conexión ODBC con el archivo dedos.xls utilizando la función `odbcConnectExcel()`.
- La tercera línea selecciona la Hoja1 de ese archivo de Excel. `Datos` pasa a ser un data.frame de R.
- La cuarta línea cierra la conexión, que ya no es necesaria.

La mejor opción importar datos de Excel si se tiene instalado excel es evitar conexiones externas.

Preparar datos en Excel

- La tabla a exportar debe estar en la esquina superior izquierda
- No puede haber más datos en esa hoja de Excel

Exportar a fichero ASCII

- Selección Fichero− >Salvar como
- Seleccionar "Formato CSV" (o personalizado)
- Seleccionar formato de exportación:
  - Carácter para delimitar cadenas de texto
  - Símbolo de los decimales
  - Símbolo para separar columnas

Importar datos en R

Leer con `read.table()`, `read.csv()` o `read.csv2()`

Licencia de este material